A Little Java in Your Hand

by Rick Grehan

The handheld market has been a difficult market for Java to infiltrate. For starters, Java execution is essentially an interpretive process, so the limited CPU of a handheld system hobbles any Java program. In addition, while bytecode programs typically take less space than equivalent programs compiled in native code, they require the attendant JVM and classfiles, which do take space.

There are other difficulties. The resources of desktop systems are largely standardized (keyboard, mouse, at least SVGA-quality display). The corresponding resources on handheld systems are (by comparison) limited and unique. Some handheld devices lack a mouse and keyboard, many have limited graphics size and no color capabilities. These limitations play out in ways that are not immediately obvious. For example, many handheld devices use a stylus as the pointing device. Without a mouse, there is no onscreen cursor, hence there are no flyover events. And the equivalent to clicking the left mouse button is tapping the display with the stylus. This places significant strain on Java's central "write once, run anywhere" pillar.

Nevertheless, to twist a trite phrase, where there's a will, there's a virtual machine.

Accentuate the Positive

Sun, with the help of the Java Community Process, has already begun establishing a range of Java "platforms." (Here, the term " platform" means a specific set of capabilities.) These platforms range from large-scale, distributed, and enterprise applications on the high-end to almost ridiculously-constrained eight-bit systems on the low end. In truth, if the JavaCard specification–which defines a Java environment for credit-card-sized smart card devices–is feasible, then surely a handheld Java is possible.

Java is aided in its move to handheld systems by the natural evolution of computer hardware. The rise in processor speed on the desktop, which has made Java's execution less and less a hindrance, will inevitably flow into the handheld world.

The remainder of this article will be an examination of existing Java development systems for handheld devices. Of course, new Java environments and tools are hitting the net all the time. We'll point you to the Web sites for these tools, but we suggest a regular prowling of newsgroups and the Internet to keep up to date on such systems.

The K Virtual Machine

Sun's offering in the handheld arena revolves around the K virtual machine (KVM). This is a component of the Java 2 Micro Edition (J2ME). (Note that J2ME defines a range of VM and Java environment capabilities; PersonalJava, for example, is part of the J2ME.)

The KVM weighs in at around 50K to 80K of object code, depending on the target platform and compilation options. (Note: 50K to 80K refers to the size of the object code only; the total memory requirement for the KVM is 128K.) It is also important to point out that the KVM was imbued with certain design features specifically for the small resources of handheld devices. For example, the KVM's built-in modularity allows features that are unnecessary in a given target implementation to be easily removed, and thereby gives implementers wide latitude in configuring the KVM for specific devices.

There is considerable overlap in the characteristics of embedded applications and handheld applications. For example, a typical embedded application is written to perform a specific function

(the "KFC" model; it does only chicken, but it does it right). Similarly, a handheld application's range of duties will usually be limited, as compared to a desktop application. Hence, it is unlikely that a handheld application will need large data types such as longs, floats, and doubles. It is equally unlikely that a handheld application will need multidimensional arrays. Such features can be omitted from a particular KVM implementation.

The KVM was written from scratch in C to optimize its memory footprint and performance. The KVM's developers took this route rather than modify an existing desktop VM so that the KVM's design would not be "infected" by artifices introduced from the desktop VM. In addition, Sun supplies tools that can further reduce codespace on the target device. For example, if a given implementation will be composed of a number of Java applications, and the developer recognizes common code among those applications, that developer can use the JavaCodeCompact tool to link the classfiles containing that common code directly into the VM.

Though the KVM does support threads, it does not rely on an OS-supplied hardware timer to generate interrupts for driving the scheduler. Instead, the virtual machine employs "bytecode counting." That is, a task-switch occurs after each "n" bytecodes are executed.

As of this writing, the GUI system that is available from Sun for the KVM is what is referred to as the "Palm overlay," which consists of tools that let you run the KVM on a Palm, and includes a simple GUI system. The PDA profile (which will describe specifics of a Palm implementation for the CLDC, Connected Limited Device Configuration) is not yet available.

The GUI is truly simple; its event-handling system is a veritable throwback to the days of the JDK 1.0.x. For example, all events having to do with touching the screen with the stylus are handled by the penDown() method. (The penDown() method must be a member of the application's class–a class that extends the Spotlet class. The Spotlet class is the foundational application class; analogous to the applet class.)

Of course, this simplification is entirely reasonable given the environment. It is unlikely that a handheld application will be composed of lots and lots of onscreen objects triggering action listeners–the situation which compelled the creation of the event delegation model for JDK 1.1x. Nevertheless, while working with the "Palm overlay" classes, it is important to keep in mind that this is only a stop-gap GUI that enables user input and output; the final shape and style of the official GUI will have to wait for the finalization of the Palm profile, which will define the specifics of Palm support for the CLDC.

Perhaps the best location online for links to KVM-related news and source code is http://www.billday.com/j2me/index.html. This Web site provides access to a surprising array of example programs for the KVM, as well as numerous supporting libraries.

Symbian/EPOC Java SDK

Devices executing Symbian's EPOC operating system tend to enjoy more powerful processors and larger memory real estate than do PalmOS devices. It can be argued that a typical EPOC device is more powerful than the Palm simply because EPOC devices strain the definition of handheld. This argument is not without merit: whereas a Psion Revo can probably be considered a handheld, the Psion netBook probably cannot. The latter is best described as a compact laptop.

(Note: The relationship between the "Symbian Platform" and " EPOC" can get confusing. Symbian defines its "Symbian Platform" as consisting of the operating system, the GUI, and applications for a particular device. The operating system is EPOC, and even Symbian admits that it is often used to refer to the entire platform. To avoid confusion, I will use "Symbian/EPOC" to refer to the platform, EPOC to refer to the OS specifically.)

Definitions of handheld aside, the Java development environment for Symbian/EPOC is at its heart, a traditional JDK 1.1x. The JVM employed in EPOC release 5 is compatible with the JDK 1.1.4. There are no missing classes or data types. And the EPOC operating system is preemptive, so the JVM does not have to rely on bytecode counting to provide task switching.

Perhaps the nicest feature of the Symbian/EPOC system is the EPOC emulator, which is part of

EPOC's Java SDK and can be downloaded from http://www.symbiandevnet.com. Install the EPOC emulator on your Windows 9x or NT system, and you can run a surprisingly complete simulation of a Symbian/EPOC device right on your desktop system.

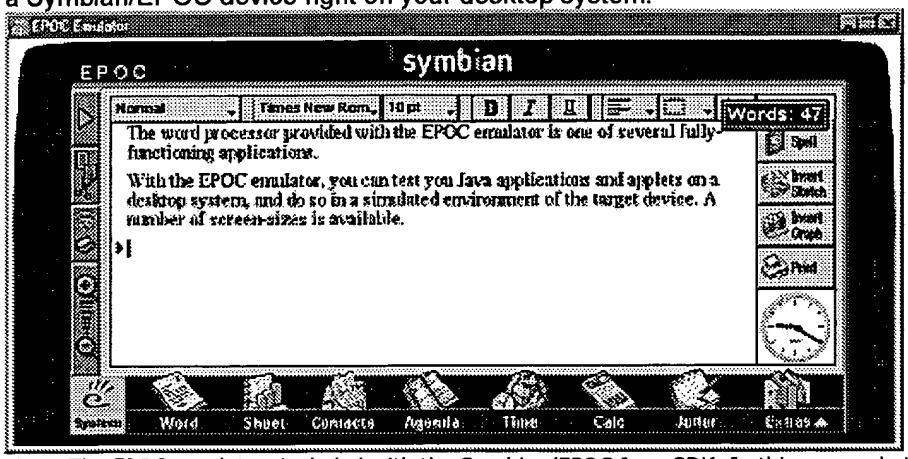

Fig 1: The EPOC emulator, included with the Symbian/EPOC Java SDK. In this screenshot, the emulator is shown running the system's word processing application, and is being used to demonstrate the transient information window available to EIKON applications.

This allows you to not only explore the applications supplied with an EPOC-based handheld device, but gives you a complete laboratory for testing your EPOC-targeted Java applications and applets. (EPOC 5 includes a web browser that is capable of running Java applets. The browser is included with the EPOC emulator, so you can experiment with the effects of running applets in the limited screen size and color depth available on Symbian/EPOC systems.)

The GUI for the EPOC OS is called EIKON. (EIKON is not only the GUI for EPOC devices, it is also a framework.) The current JDK for EPOC implements AWT atop EPOC, and this arrangement has pros and cons.

On the one hand, because the developer "sees" AWT (and not EIKON), porting a Java application already running on the desktop (and which uses AWT) involves wrestling only issues of screen size and user-interface mechanics (for example, how a stylus-based user interface behaves as compared to a mouse-based interface). The developer doesn't have to rewrite code to translate AWT constructs into their EIKON equivalents. This benefit is obviously significant to a programmer new to the EPOC platform.

On the other hand, there is functionality in EIKON that the programmer using AWT will never see. As the EPOC version of AWT is implemented "atop" EIKON, a translation layer is in place, and that layer represents a performance loss. That is, rather than call EIKON directly, the Java application first calls AWT and AWT calls EIKON on the application's behalf. In addition, some EIKON graphical and non-graphical capabilities are simply not available to programmers through AWT.

Symbian has taken some steps to correct this. A number of classes are available for download from the Web site that--via native method calls--give some EIKON capabilities to Java programs. For example, the TaskSwitch Java Native Interface includes several native methods that expose EIKON functions related to task switching. (Here, a task refers to an executing application; " taskswitching" refers to making a particular application to current foreground application.) The image in figure 1, which is a screenshot of the EPOC emulator running on a Windows desktop, shows one such feature provided in the TaskSwitch interface and otherwise unavailable to Java applications. This is a temporary information window that appears in the upper right-hand corner of the screen, lingering for several seconds before disappearing. In this instance, the information window is used to report the number of words in the word processing file. (It appears in response to a word-count request from the Tools menu entry in the word processing application.)

At the time of this writing, a new release of the Symbian/EPOC platform (version 6.0) was just being made available. As with EPOC 5, SDKs for the C/C++, Java, and OPL (a programming language from Symbian's Psion roots) were included. In a kind of mimicry of Sun's CLDC and MIDP, Symbian has defined "device family reference designs" (DFRDs). There are currently three such reference

designs:

- Quartz, for quarter-VGA screen-sized, pen-based systems that are described as "tablet-like",
  - Crystal, for half-VGA screen-sized, keyboard based systems, and
  - Pearl, for smartphone systems.

More information is available from the Symbian Developer Web site (mentioned above).

Wabasoft's Waba

Waba is Java, and it is not Java. That's an oblique way of saying Waba is an implementation that is not standard Java, and may therefore not make any claims to being Java. Paradoxically, this is probably its greatest strength.

In essence, the people at Wabasoft conducted their own personal community process some time ago, and produced their own specification for a CLDC Java. The result is a Java-like package that is best described as: "Simple, but not too simple" (with apologies to Einstein). Waba is a proper subset of Java, and therefore you can use standard Java development tools to write Waba programs. Waba programs can run as Java applets and applications, but the converse is not true (a Java applet or application that uses the standard Java classfiles cannot run on the Waba VM).

Currently, Waba is available for the Palm OS, Windows CE, and Windows 9x/NT. The desktop (9x/NT) version of Waba is accompanied by a set of "bridge classes" that provide a translation layer between the Win32 GUI and the Waba GUI. In short, with the bridge classes, you can develop and execute a Waba application on a Windows desktop machine and thereby test it on your desktop before downloading it to the handheld. (According to the Wabasoft site, an independent party has ported the Waba tools to Linux and Solaris.)

Waba consists of six packages, as listed below:

waba.fx - This package covers the drawing primitives, and includes classes for rendering images, drawing fonts, and managing colors. In addition, this class handles sounds.

waba.io - The waba.io package includes classes for handling files, the serial port, and sockets. The Catalog class in this package includes methods for interacting with the database systems on PalmOS and Windows CE.

waba.ui - This package includes the classes that manage the user interface (e.g., Window, TabBar, Edit fields, etc.).

waba.sys - This package includes system-related classes for time and data type conversion, as well as VM-specific classes.

waba.util - This package's sole member is Vector, which gives the developer access to an arbitrarily-long storage class.

waba.lang - This package is actually not part of the SDK. It includes useful classes–String, StringBuffer, and Object–which are "Waba equivalents" to the java.lang package. At development time, you use the java.lang equivalents, and the runtime encounters references to the java.lang classes, it maps those references to their equivalents in waba.lang.

An example of a Waba application appears in the following listing. The program implements a simple stopwatch with one-second accuracy, and illustrates not only the simple event-handling structure of a Waba application, but shows the use of a timer object to generate timer events in one-second intervals.

LISTING 1.

```
import waba.ui.*;
import waba.fx.*;
import waba.sys.*;
```

```
public class WabaStopWatch extends MainWindow
{
// Define buttons
Button startButton;
Button stopButton;
Button clearButton;
Button exitButton;
// Define output display field
Label timeLabel;
// Define fields
int hours;
int minutes;
int seconds;
// Graphics object for output
Graphics g;
// Timer for the stopwatch
Timer timer;
// Indicate timer started/stopped
boolean running;
public WabaStopWatch()
{
// Initialize fields
hours=0;
minutes=0;
seconds=0;
// Make a graphics object
g = new Graphics(this);
// Create and add the buttons
startButton = new Button("START");
startButton.setRect(10,100,35,15);
add(startButton);
stopButton = new Button("STOP");
stopButton.setRect(50,100,30,15);
add(stopButton);
clearButton = new Button("CLR");
clearButton.setRect(85,100,23,15);
add(clearButton);
exitButton = new Button("EXIT");
exitButton.setRect(110,100,25,15);
add(exitButton);
timeLabel = new Label("TIME:");
timeLabel.setRect(20,60,35,20);
add(timeLabel);
// Set the timer event
if (timer == null)
{
// update every second
timer = addTimer(1000);
}
// Not running yet
running=false;
// Initial display
dodisplay();
}
//
// onEvent()
// Called each time an event occurs
public void onEvent(Event event)
{
// See if this was a timer event.
```

```
// If so, update the timer if the
// watch is running
if(event.type == ControlEvent.TIMER)
{
if(running == false) return;
seconds++;
if(seconds==60)
{ seconds=0;
minutes++;
if(minutes==60)
{ minutes=0;
hours++;
}
}
dodisplay(); // Display update
}
if(event.type==ControlEvent.PRESSED)
{
if(event.target == startButton)
running=true;
if(event.target == stopButton)
running=false;
if(event.target == clearButton)
{ running=false; // Implicit stop
hours=0;
minutes=0;
seconds=0;
dodisplay();
}
if(event.target == exitButton)
exit(0);
}
}
public void dodisplay()
{
String ostring;
// Construct output string
ostring = new String("");
if(hours<10) ostring="0";
ostring += Convert.toString(hours) + ":";
if(minutes<10) ostring+="0";
ostring += Convert.toString(minutes) + ":";
if(seconds<10) ostring+="0";
ostring += Convert.toString(seconds);
// Clear the field
g.setColor(255,255,255);
g.fillRect(50,64,50,20);
// Display the string
g.setColor(0,0,0);
g.drawText(ostring,50,64);
}
}
```

Fig 2: The Waba stopwatch, shown here running on the desktop, is using a standard Java VM. The stopwatch is accurate to the second, and includes basic capabilities; starting, stopping, and clearing the stopwatch.

Figure 2 shows this application running on the desktop. The Waba development kit includes tools for converting the application into the format necessary to download it to the Palm (or Windows CE) device. After creating a simple bitmap icon (using Windows Paint), I ran the tools to create a Palm application, and downloaded it to the Palm emulator (POSE) on my system. Figure 3 shows the application with its icon in place on the simulated Palm device. Figure 4 shows the same stopwatch application running on the Palm.

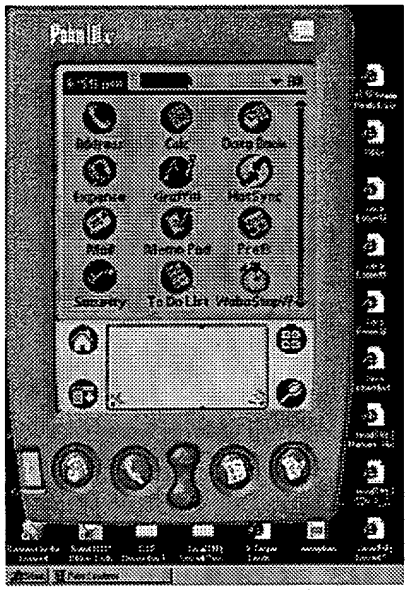
Fig 3: The Waba stopwatch, shown here ported to the Palm device. It appears on the Palm's home screen as a small watch icon (easily created using Windows Paint). Tools provided with the Waba toolkit bundle the icon automatically, and create the necessary files for downloading to the Palm.

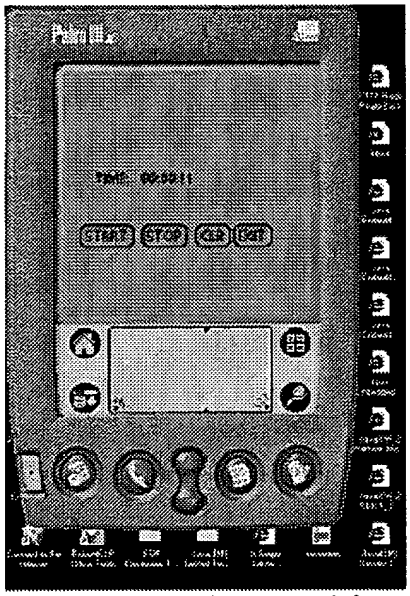

Fig 4: The same Waba stopwatch from figure 2, now running on the Palm. Note that no recompilation was necessary from figure 2 to figure 4.

All the software and tools you need for developing and deploying Waba applications are available from the Wabasoft site at http://www.wabasoft.com. (Even the Waba virtual machine's source code is available. Wabasoft indicates that having the virtual machine's source is the avenue for adding native code capabilities to an application.) The tools are governed by the GNU license.

Grab Some Java

Anyone interested in Java development for handheld systems has several choices to explore. And there are even more than we've mentioned. For example, IBM's Visual Age Micro Edition (VAME) for Java allows for development of Java applications for the PalmOS. The VAME virtual machine (referred to as the J9 VM) is modular, and therefore can be configured to suit a particular application. You can download IBM's VAME from the company's embedded technologies Web site at http://www.embedded.oti.com.

As this article was going to press, another company, Aromasoft, had just ported its TeaPot OS and Java development tools to the Compaq iPAQ PocketPC.

Though Java's place in handheld systems is far from secure, it is comforting to see the development options already available. Best of all, you need not own a handheld to try your hand–no pun intended–at handheld development. The sophistication of simulations provided by the EPOC emulator and Waba's bridge classes give your Java applications a remarkably accurate experience of the target environment right on the desktop.

As these tools improve, as the ability to tailor-make a VM for a specific device appears, and as the CPU power and available memory on handheld devices rise and rise, few excuses remain for *not* putting Java in the palm of your hand.

Go back